Algorithms and Data Structures 2
CS 1501

Spring 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming deadlines:
  - Lab 10 due on 4/8
  - Homework 11 due on 4/11
  - Assignment 3 and 4 due on 4/18
    - Used to be one assignment

Previous lecture …

- Network Bottleneck Finding Problem
  - Ford-Fulkerson Max-Flow Framework
    - augmenting path
    - backwards edges
  - Edmonds-Karp algorithm
    - using BFS to find augmenting paths

CourseMIRROR Reflections (most confusing)

- why is the max of the back edge the numerator of the forward edge?
- How/why we are able to take the backwards paths whenever the graph was originally directed
- Can any path from s to t be an augmenting path?
- How are bottlenecks defined? Are they flow edges that saturate early?
- Dijkstra algorithm but I will go over it by using past lectures
- Flow was mostly clear but doing another example would be helpful
- I was confused about how the runtime is calculated and what role maximum flow plays in it
- Going over backwards paths one more time and showing how they reroute the existing data flow would be helpful
- Edmonds Karp was most confusing.
- the ford example with 2000 iterations was confusing

CourseMIRROR Reflections (most interesting)

- How the max flow value is directly involved in the runtime of Ford-Fulkerson
- djikstras algorithm for shortest path explanation
- The data flow problem
- Residual graph
- The concept of using back edges
- Optimizing max flow through use of back edges
- adding backwards edges
- How backwards edges can help correct mistakes
- I found it interesting how BFS can help find any augmenting path
- The bottleneck problem in general was most interesting.
- ford fulkerson

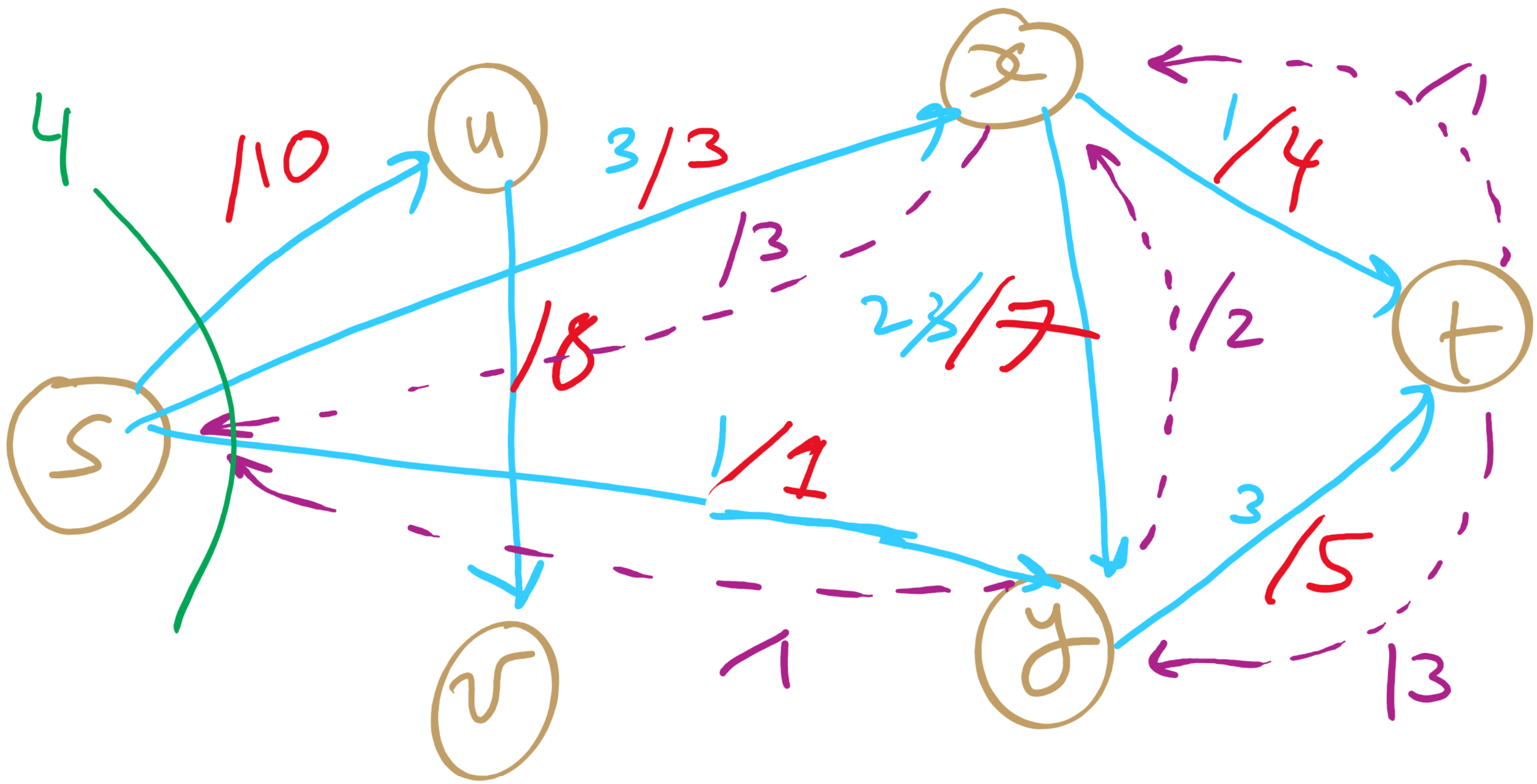
Problem of the Day: Finding Network Bottlenecks

- Let's assume that we want to send a large file from point A to point B over a computer network as fast as possible over multiple network links if needed
- Input:
  - A computer network
    - Network nodes and links
    - Links are labeled by link capacity in Mbps
  - Starting node and destination node
- Output:
  - The maximum network speed possible for sending a file from source to destination

But our flow graph is weighted...

- Edmonds-Karp only uses BFS
  - Used to find spanning trees and shortest paths for *unweighted* graphs
  - Why do we not use some measure of priority to find augmenting paths?

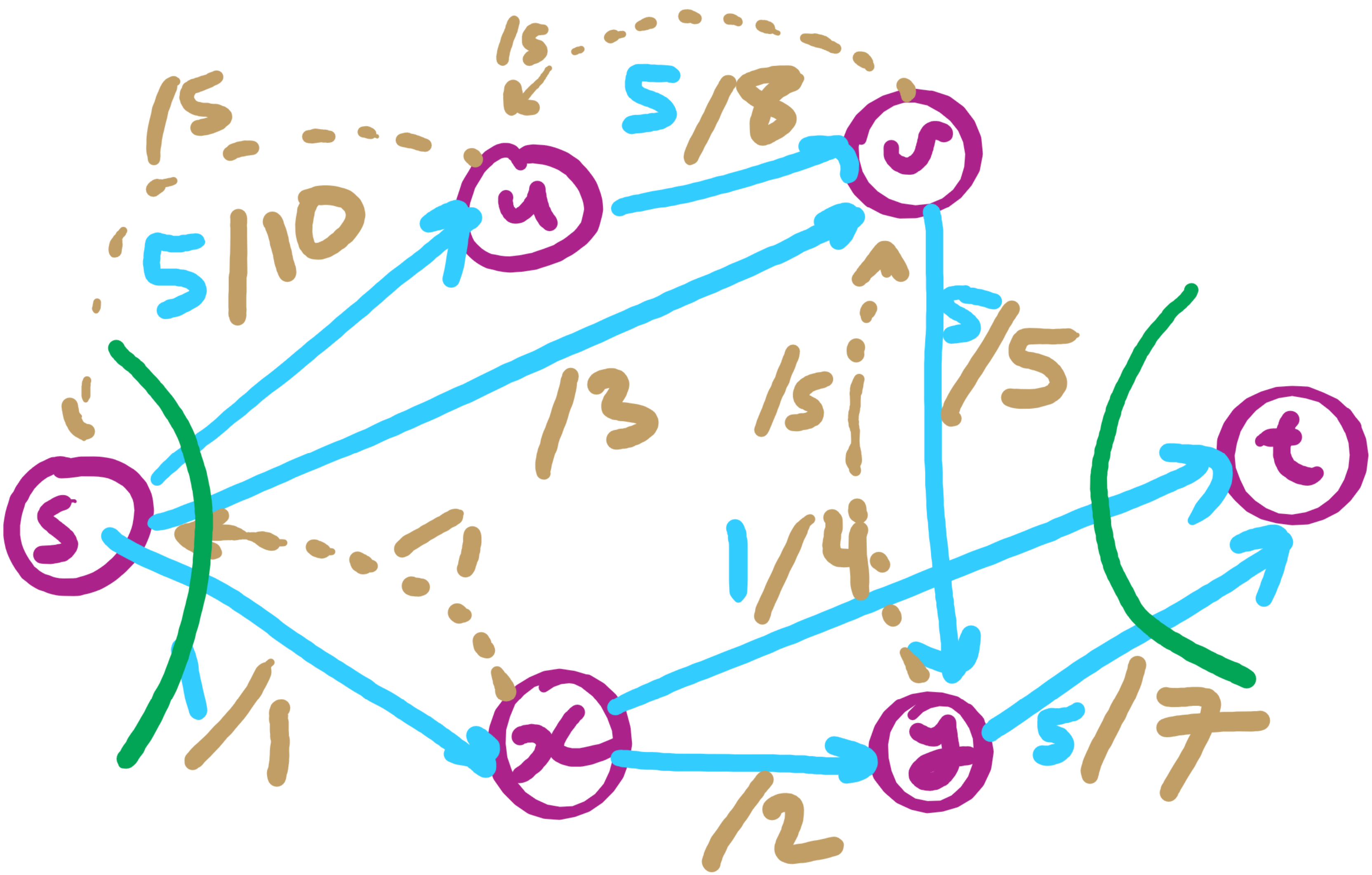
PFS Example 1



$s \rightarrow x \rightarrow y \rightarrow t$

$s \rightarrow y \rightarrow x \rightarrow t$

PFS Example 2



Flow edge implementation

- For each edge, we need to store:
  - Start point, the from vertex
  - End point, the to vertex
  - Capacity
  - Flow
  - Residual capacities
    - For forwards and backwards edges

FlowEdge.java

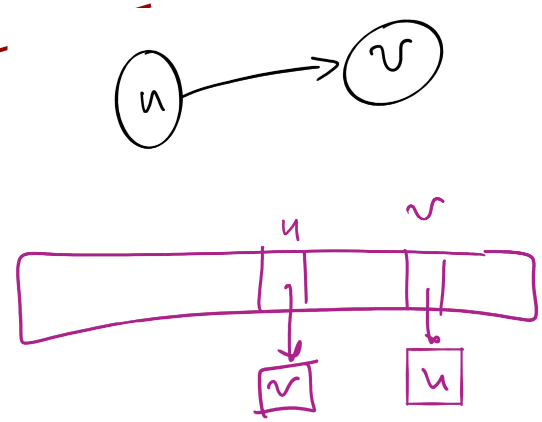
```
public class FlowEdge {
   private final int v;               // from
   private final int w;               // to
   private final double capacity;    // capacity
   private double flow;               // flow
     …
     public double residualCapacityTo(int vertex) {
      if       (vertex == v) return flow;
      else if (vertex == w) return capacity - flow;
      else throw new
       IllegalArgumentException("Illegal endpoint");
   }
   …
}
```

BFS search for an augmenting path (pseudocode)

```
edgeTo = [|V|]
marked = [|V|]
Queue q
q.enqueue(s)
marked[s] = true
while !q.isEmpty():
    v = q.dequeue()
    for each (v, w) in AdjList[v]:
        if residualCapacity(v, w) > 0:
            if !marked[w]:
                edgeTo[w] = v;
                marked[w] = true;
                q.enqueue(w);
```

Each FlowEdge object is stored in the adjacency list twice:

Once for its forward edge
Once for its backward edge



Value of maxflow

- Add up the flow increments in each iteration of Ford-Fulkerson
- Add up the edge **flows** out of source
- Add up the edge **flows** of the out of source

Follow-up Problem

- So, now we found the bottleneck *value*, but which edges define the found bottleneck?

    - *Why would you want to know those bottleneck edges?*

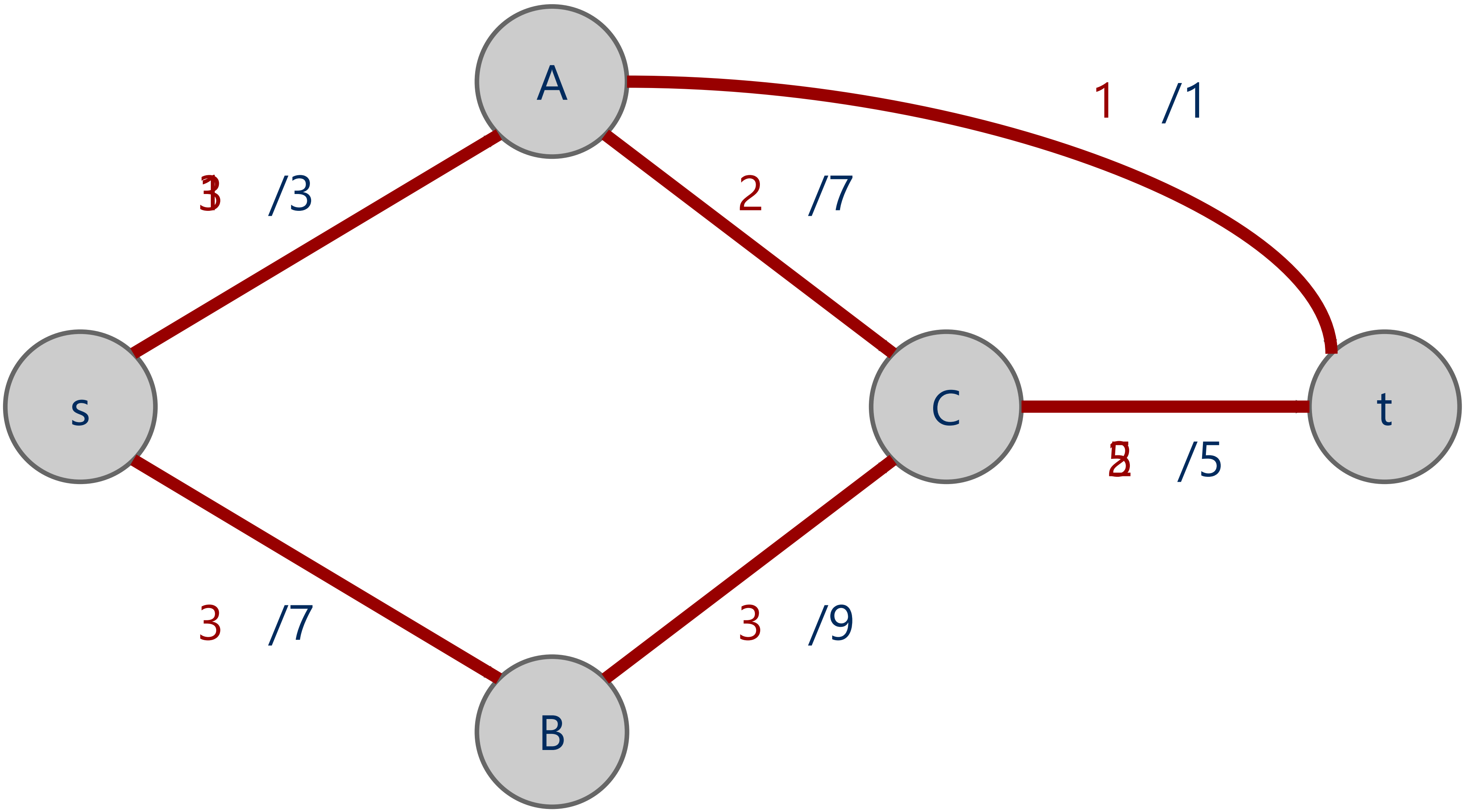
Let's separate the graph

- An st-cut on G is a set of edges in G that, if removed, will partition the vertices of G into two disjoint sets
  - One contains s
  - One contains t
- May be many st-cuts for a given graph
- Let’s focus on finding the minimum st-cut
  - The st-cut with the smallest capacity
  - May not be unique

How do we find the min st-cut?

- We could examine residual graphs
  - Specifically, try and allocate flow in the graph until we get to a residual graph with no existing augmenting paths
    - A set of saturated edges will make a minimum st-cut

Min cut example



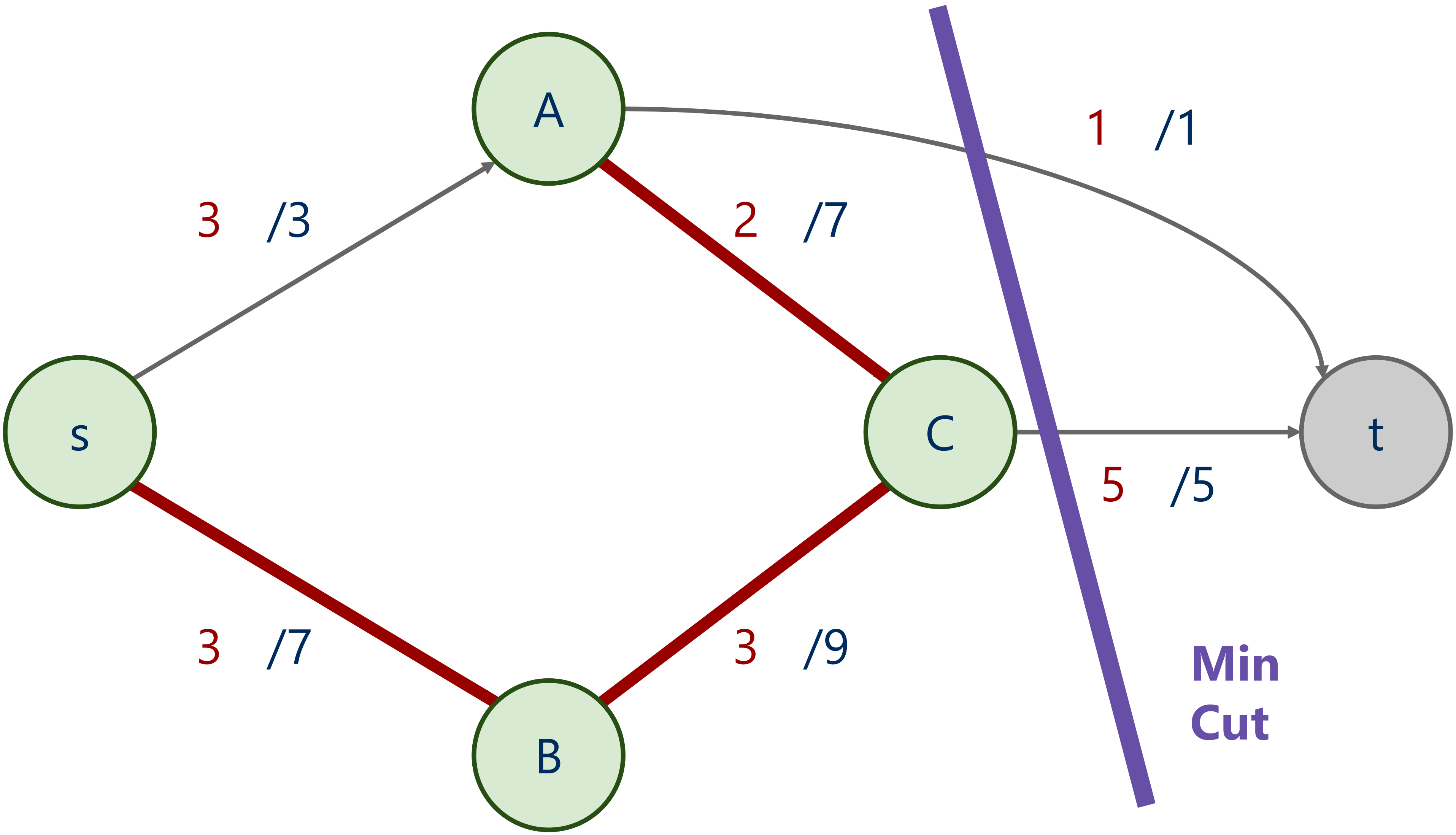
Max flow == min cut

- A special case of duality
  - I.e., you can look at an optimization problem from two angles
    - In this case to find the maximum flow or minimum cut
  - In general, dual problems do not have to have equal solutions
    - The differences in solutions to the two ways of looking at the problem is referred to as the *duality gap*
      - If the duality gap = 0, strong duality holds
        - Max flow/min cut uphold strong duality
      - If the duality gap > 0, weak duality holds

Determining a minimum st-cut

- First, run Ford Fulkerson to produce a residual graph with no further augmenting paths
- The last attempt to find an augmenting path will visit every vertex reachable from s
    - Edges with only one endpoint in this set comprise a minimum st-cut

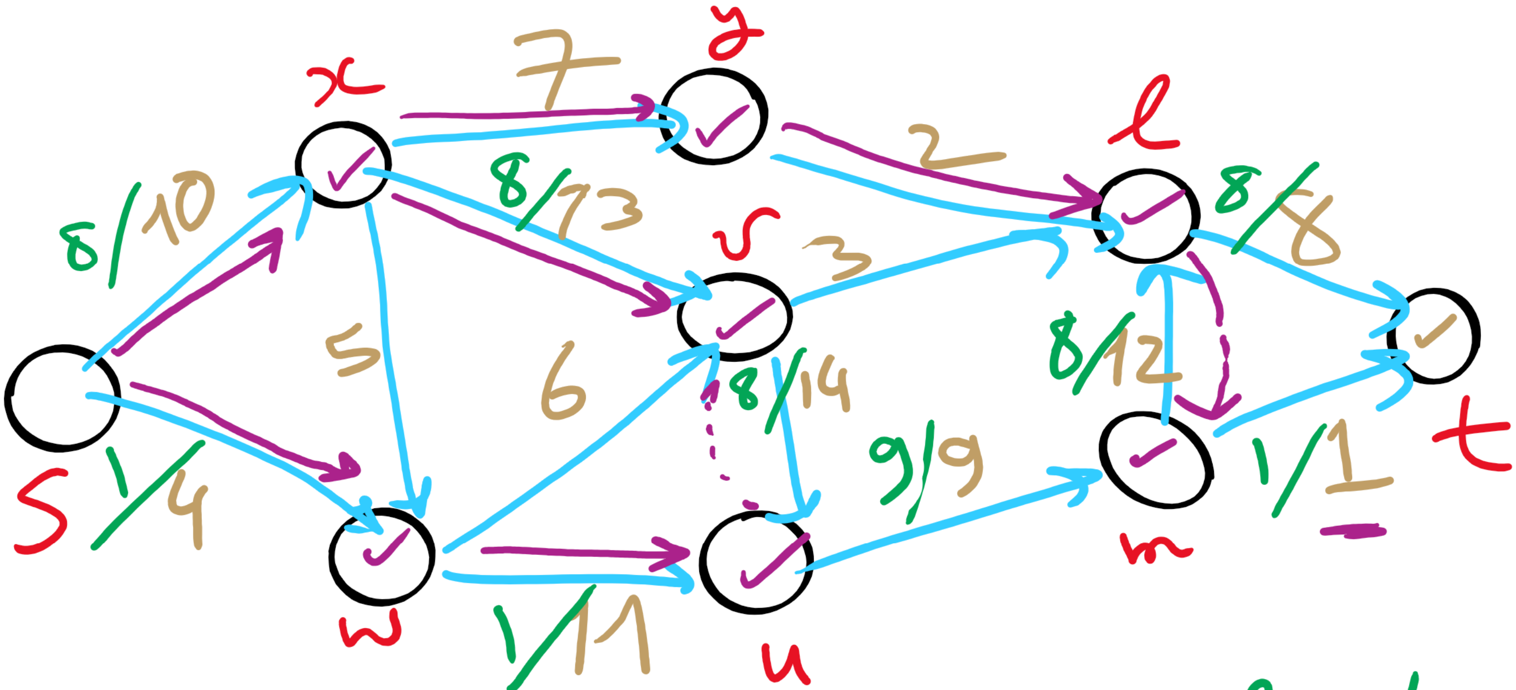
Determining the min cut



Max flow / min cut on unweighted graphs

- Is it possible?
- How would we measure the Max flow / min cut?
- What would an algorithm to solve this problem look like?

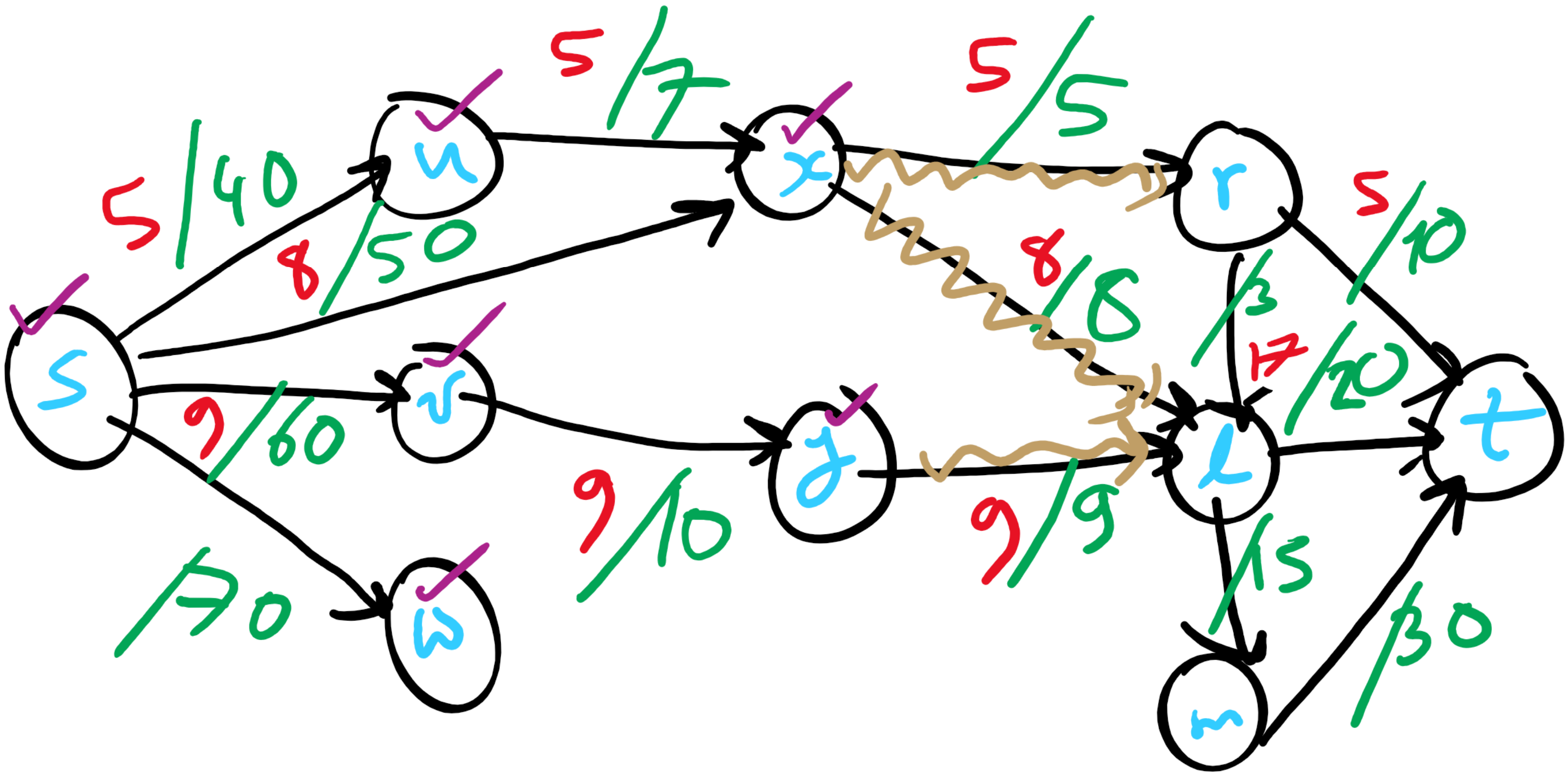
Min st-cut Example 1



s → x → v → u → m → l → t
s → w → u → m → t

l → t
m → t } a min-cut

Min st-cut Example 2

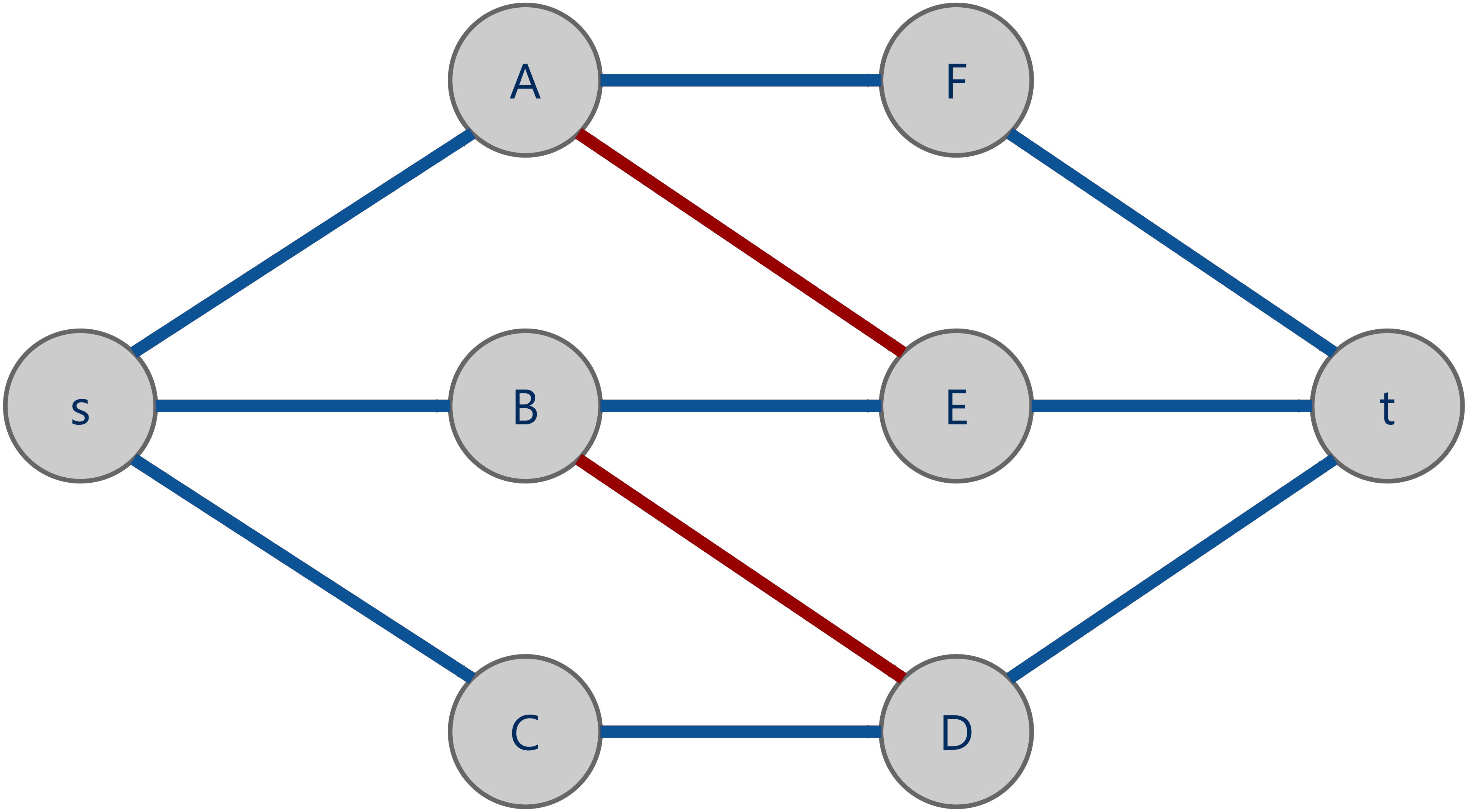


$s \to u \to x \to r \to t$

$s \to v \to y \to l \to t$

$s \to x \to l \to t$

Unweighted network flow



Graph Algorithms Runtime

BFS/DFS Connected Components Articulation Points shortest Pathes based on # hops	$\Theta(e+v)$ Adj. lists	$\Theta(v^2)$ Adj Matrix		
Prim's MST	$\Theta(v^3)$ naive	$\Theta(v^2)$ Best Edge	$\Theta(e \log e)$ PQ (Lazy)	$\Theta(e \log v)$ Indexable PQ (Eager)
Kruskal's MST	$\Theta(e(e+v))$ BFS/DFS for cycle detection	$\Theta(e \log e)$ UF weighted Trees		
Dijkstra's shortest pathes	$\Theta(v^2)$ Distance array	$\Theta(e \log v)$ Indexable PQ		
Ford-Fulkerson	$\Theta(\lvert f\rvert \cdot (e+v))$			
Edmonds-Karp (Max Flow & Min Cut)	$\Theta(e^2 \cdot v)$			

Problem of the Day: Integer multiplication

- Input: two *n-bit* integers *x* and *y*
  - n can be really large!
    - e.g., 2048 bits
    - Why do we need such big numbers?
  - The input size is $\theta(n)$
  - The values of *x* and *y* are exponential in the input size!
- Output:
  - x * y

Yeah, but the processor has a MUL instruction

- Assuming x86
- Given two 32-bit integers, MUL will produce a 64-bit integer in a few cycles
- What about when we need to multiply large ints?
  - VERY large ints?
    - RSA keys should be 2048 bits
  - Back to grade school…

Gradeschool algorithm on binary numbers

```
        10100000100
x       11000101111
-------------------
        10100000100
       101000001000
      1010000010000
     10100000100000
    000000000000000
   1010000010000000
  00000000000000000
 000000000000000000
0000000000000000000
10100000100000000000
101000001000000000000
-------------------
111110000001110111100
```

OK, I'm guessing we all knew that...

- What is the runtime of this multiplication algorithm?
  - For 2 n-digit numbers:
    - n^2

How can we improve our runtime?

- Let's try to divide and conquer:
  - Break our n-bit integers in half:
    - $x = 1001011011001000$, $n = 16$
    - Let the high-order bits be $x_H = 10010110$
    - Let the low-order bits be $x_L = 11001000$
    - $x = 2^{n/2}x_H + x_L$
    - Do the same for y
    - $x * y = (2^{n/2}x_H + x_L) * (2^{n/2}y_H + y_L)$
    - $x * y = 2^n x_H y_H + 2^{n/2}(x_H y_L + x_L y_H) + x_L y_L$

So what does this mean?

Multiplying
2 4-bit ⇒ 8-bit
2 n/2-bit ⇒ n-bit
2 n-bit = 2n-bit

4 multiplications of n/2 bit integers

3 additions of n-bit integers

$$2^n x_H y_H + 2^{n/2}(x_H y_L + x_L y_H) + x_L y_L$$

A couple shifts of up to n positions

Actually 16 multiplications of n/4 bit integers (plus additions/shifts)

Actually 64 multiplications of n/8 bit integers (plus additions/shifts)

...

Karatsuba's algorithm

- By reducing the number of recursive calls (subproblems), we can improve the runtime
- $x * y = 2^n x_H y_H + 2^{n/2}(x_H y_L + x_L y_H) + x_L y_L$

  M1 M2 M3 M4

- We don't actually need to do both M2 and M3
  - We just need the sum of M2 and M3
    - If we can find this sum using only 1 multiplication, we decrease the number of recursive calls and hence improve our runtime

Karatsuba craziness

- $M1 = x_h y_h$; $M2 = x_h y_l$; $M3 = x_l y_h$; $M4 = x_l y_l$;
- The sum of all of them can be expressed as a single mult:
  - $M1 + M2 + M3 + M4$
  - $= x_h y_h + x_h y_l + x_l y_h + x_l y_l$
  - $= (x_h + x_l) * (y_h + y_l)$
- Lets call this single multiplication M5:
  - $M5 = (x_h + x_l) * (y_h + y_l) = M1 + M2 + M3 + M4$
- Hence, $M5 - M1 - M4 = M2 + M3$
- So: $x * y = 2^n M1 + 2^{n/2}(M5 - M1 - M4) + M4$
  - Only 3 multiplications required!
  - At the cost of 2 more additions, and 2 subtractions

Karatsuba Example

$$x = \underbrace{97754}_{x_H}\underbrace{01238}_{x_L}$$

$$y = \underbrace{14597}_{y_H}\underbrace{01003}_{y_L}$$

$$\boxed{n == 10}$$

$$M_1 = x_H \cdot y_H$$

$$M_2 = x_L \cdot y_L$$

$$M_5 = (x_L + x_H)(y_L + y_H)$$

$x * y =$ M_1 shifted left by n positions

$+ (M_5 - M_1 - M_2)$ shifted left by $n/2$ positions

$+ M_2$

Large integer multiplication in practice

- Can use a hybrid algorithm of grade school for large operands, Karatsuba's algorithm for VERY large operands
  - Why are we still bothering with grade school at all?

Is this the best we can do?

- The Schönhage–Strassen algorithm
  - Uses Fast Fourier transforms to achieve better asymptotic runtime
    - O(n log n log log n)
    - Fastest asymptotic runtime known from 1971-2007
      - Required n to be astronomical to achieve practical improvements to runtime
        - Numbers beyond $2^{2^{15}}$ to $2^{2^{17}}$
- Fürer was able to achieve even better asymptotic runtime in 2007
  - n log n $2^{O(\log^* n)}$
  - No practical difference for realistic values of n

$\log^* n$ = # times we can compute log n until the result ≤ 1

Please submit your reflections by using the CourseMIRROR App

If you are having a problem with CourseMIRROR, please send an email to coursemirror.development@gmail.com